

期末作业

期末作业

1. 游戏引擎框架初步分析
2. 外挂程序初步分析
 - 2.1 hack.exe反反调试与内存dump
 - 2.2 hack_dump_FIX.exe逆向分析
3. 游戏内存初步分析
 - 2.1 用CE进行游戏内容的简单分析
 - 2.2 UE4.22游戏指针链分析
4. 外挂逻辑分析
5. 总结

- 我参考并学习的article链接

1. [UE4逆向笔记之GWorld GName GameInstance](#)
2. [Quick look around VMP 3.x - Part 1 : Unpacking](#)
3. [Trouver l'OEP](#)

- 作业环境

OS	Arch
Microsoft Windows 10	Intel64

- 使用工具或插件

Tools	Version
Detect It Easy	v3.01
Cheat Engine	v7.5
IDA Pro	v7.5
x64dbg	Jun 15 2022, 19:59:36
Scylla x64	v0.9.8
ScyllaHide	v1.4.750
ApiBreak	v0.5

1. 游戏引擎框架初步分析

在分析外挂之前，先了解一下目标游戏的框架背景。

Detect It Easy v3.01

文件名: D:/Learning/2023project/TLearning/菁英班PC端期末作业/game/ShooterGame/Binaries/Win64/ShooterClient.exe

文件类型: PE64 入口点: 00000001420f4540 基址: 0000000140000000 内存映射: ...

区块: 0008 导出: TimeStamp: 2020-06-12 16:52:46 SizeOfImage: 032b1000 资源: .NET: TLS: 覆盖: 显示: 版本: MIME: Hash: 字符串: 信息熵: Hex:

扫描: Detect It Easy (DiE) 字节序: LE 模式: 64 架构: AMD64 类型: GUI

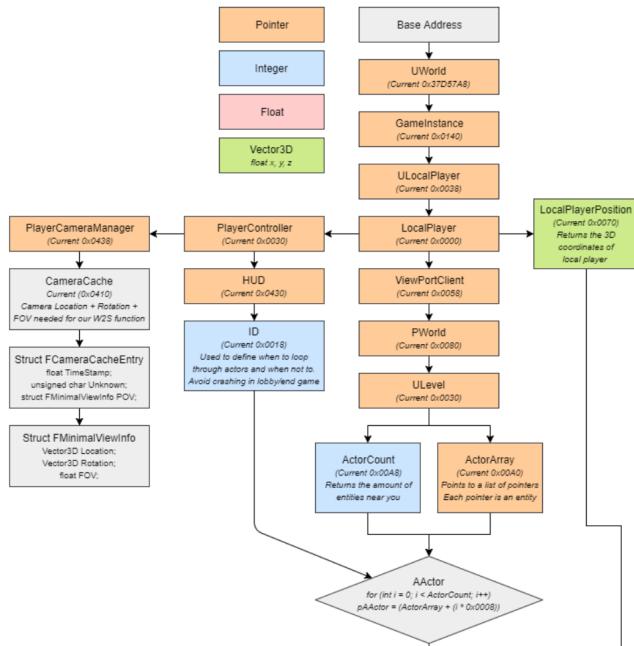
compiler: Microsoft Visual C++ (2017 v.15.9)[-] S ?
linker: Microsoft Linker(14.16, Visual Studio 2017 15.9*)[GUI64] S ?

签名: 深度扫描: 扫描: 选项: 关于: 退出:

100% 日志: 270 毫秒

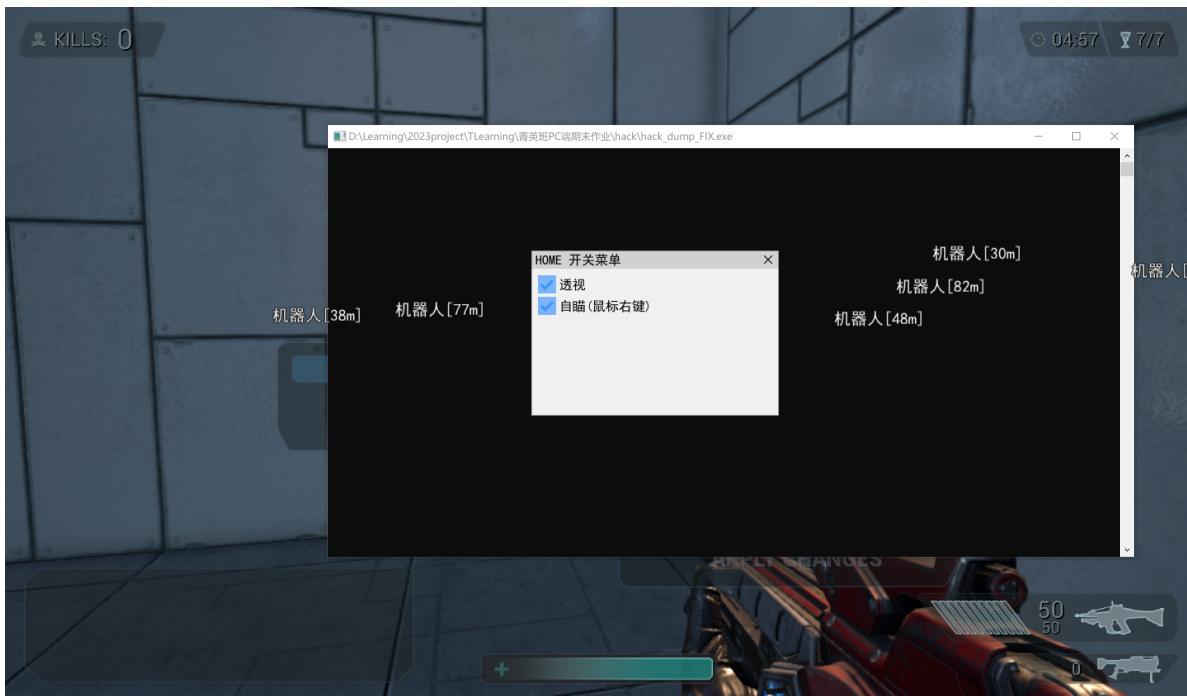
IDB View-A Occurrences of: +UE Address Function Instruction
.text:00000001405701C0 lea rax, aUe4Release422 ; "++UE4+Release-4.22"
.text:00000001405701E0 lea rax, aUe4Release422C ; "++UE4+Release-4.22-CL-0"

直接查看游戏属性，发现基于虚幻引擎Unreal Engine，版本是4.22，故后续分析游戏对象的类继承关系必须详细了解UE4版本的下的类继承关系结构和导图。



如上图，从UWORLD到GameInstance到ULocalPlayer到LocalPlayer再到PlayerController乃至角色对象AAActor，UE4中有着层次非常丰富的类继承关系，所以如果我们要找到游戏中指定的对象属性，需要找到一条很长的**实例对象指针链**才能访问该对象的属性。

同时ULevel下的**ActorCount**和**ActorArray**存放着所有角色对象，可以通过它们遍历游戏场景中的所有角色。



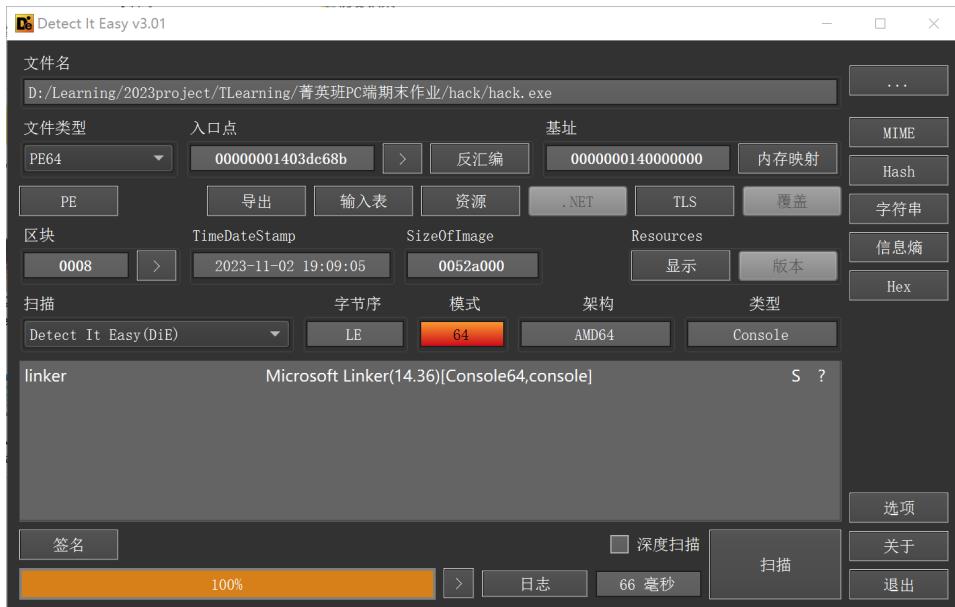
进入游戏，分析外挂，**HOME** 键可以控制外挂菜单的开关，单击菜单可以控制外挂的开关。

1. 透视：相对位置在屏幕视野内的机器人目标是可以被标记出来的，格式为 机器人 [%dm] 。
2. 自瞄：单击鼠标右键，准星会自动吸附距离最近的一个机器人的身体。

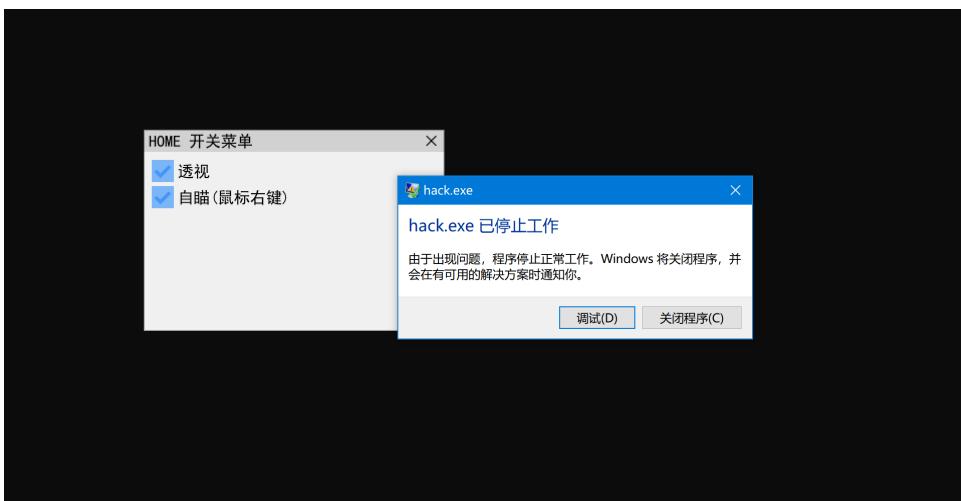
2. 外挂程序初步分析

2.1 hack.exe反反调试与内存dump

查看hack.exe程序的情况，似乎没有扫出保护层。



用x64dbg打开，直接F9往后调试，当看到 EntryPoint 时就已经觉得不对劲了，因为是熟悉的“push imm, call”结构，说明存在保护壳。继续向下调试，hack.exe停止工作，说明还存在反调试，所以先破一下反调试。

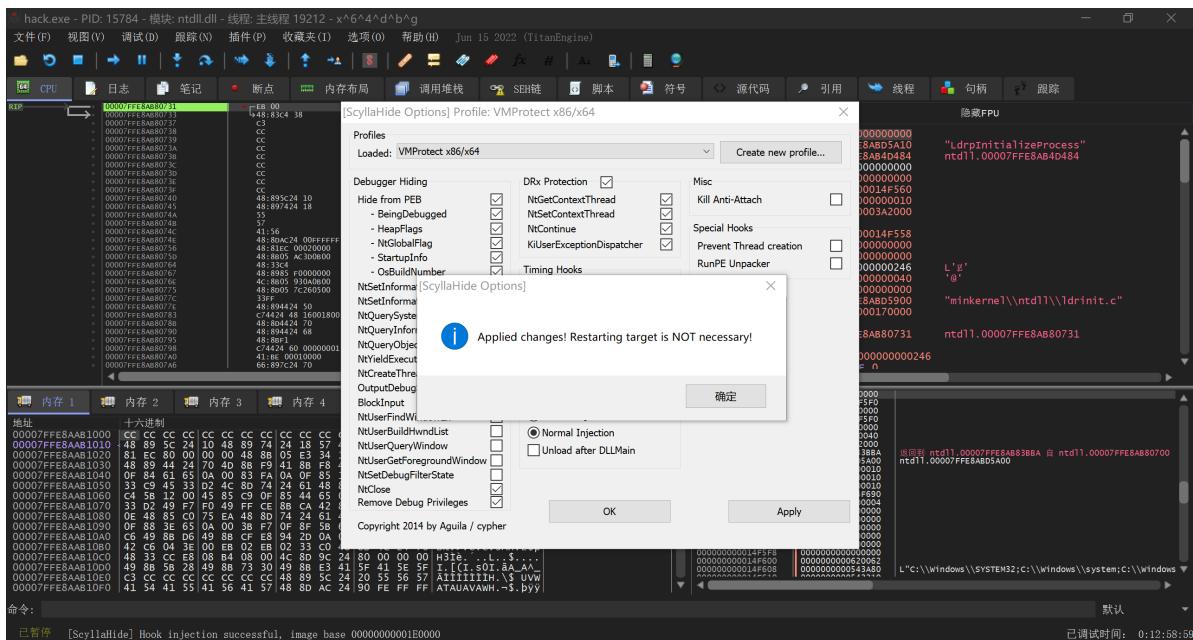


hack.exe的入口点如下图，这是VMProtect虚拟化保护的常见特征，下面先绕过反调试，再通过进入程序内部后查看函数调用栈回溯来判断OEP。

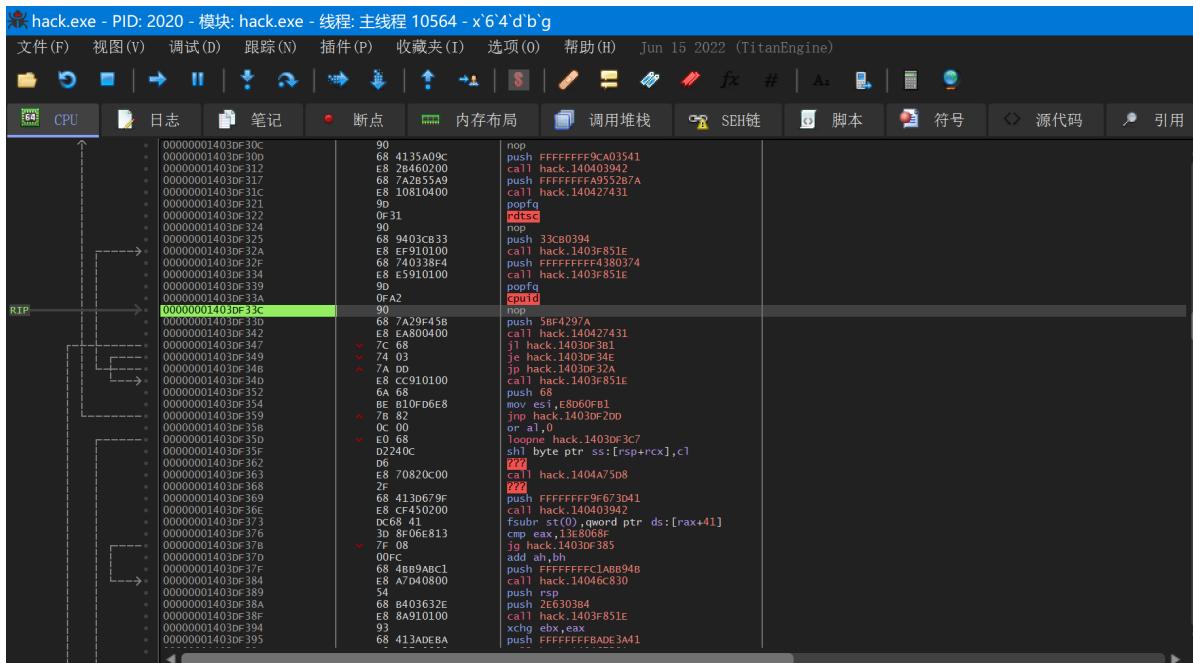
Note that the entrypoint looks like a VMProtect virtualized routine, and in fact, it is ! In the past (version 2.x), VMP unpacking routine were not virtualized (like in this video), so it was easy to rush and breakpoint the jump to OEP (push and ret).

→ 00605EFF G8 A3E918A4 push A418E9A3
 → 00605F04 E8 EEDB2500 call mcc.packed.863AF7
 → 00605F09 0FC8 bswap eax
 → 00605F0B C1C8 02 ror eax,2
 → 00605F0E 66:3BE8 cmp bp,bx
 → 00605F11 8D80 4439E25E lea eax,dword ptr ds:[eax+5EE23944]
 → 00605F17 F5 cmc
 → 00605F18 35 E35E0E5D xor eax,SD0E5EE3
 → 00605F1D 8D80 2C485826 lea eax,dword ptr ds:[eax+2658482C]

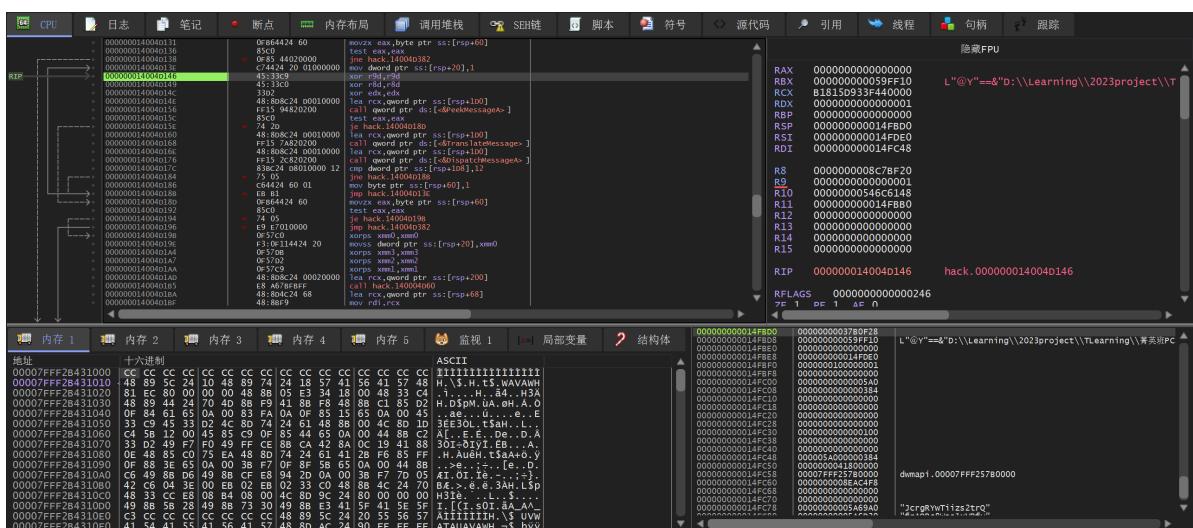
使用ScyllaHide绕过vmp的反调试。（有工具干嘛自己动手~



进入EntryPoint后连续按3次F9被nop断下后，即可绕过反调试。



在之后单步运行，其实这里有两种方法，一种是下图我在程序读入窗口的交互信息时断下，可以看到明显的PeekMessageA、TranslateMessage、DispatchMessageA等窗口交互库的调用，我在这些的栈中检查函数调用链，在倒数第二个call调用中找到了疑似main函数的调用。



调用链为 call [0x1400525A4] -> call [0x14003480] -> call [0x14004CD10]

```

0000000014FDD0 0000000000000000 "写K"=="ALLUSERSPROFILE=C:\\\\ProgramData"
0000000014FDD8 0000000000000000 返回到 hack.0000000140003629 自 hack.000000014004CD10
0000000014FDE0 0000000000000000
0000000014FDE8 0000000000004C5BA0
0000000014FDF0 0000000000000000
0000000014FDF8 0000000000000000
0000000014FE00 0000000000000000
0000000014FE08 0000000000000000
0000000014FE10 0000000000000000
0000000014FE18 0000000000000000
0000000014FE20 0000000000000000
0000000014FE28 0000000000000000
0000000014FE30 0000000000000000
0000000014FE38 0000000000000000
0000000014FE40 0000000000000000
0000000014FE48 0000000000000000
0000000014FE50 0000000000000000
0000000014FE58 0000000000000000
0000000014FE60 0000000000000000
0000000014FE68 0000000000000000
0000000014FE70 0000000000000000
0000000014FE78 0000000000000000
0000000014FE80 0000000000000000
0000000014FE88 0000000000000000
0000000014FE90 0000000000000000
0000000014FE98 0000000000000000
0000000014FEAO 0000000000000000
0000000014FEA8 0000000000000000
0000000014FEB0 0000000000000000
0000000014FEB8 0000000000000000
0000000014FEC0 0000000000000000
0000000014FEC8 0000000000000000
0000000014FED0 0000000000000000
0000000014FEE8 0000000000000000
0000000014FEF0 0000000000000000
0000000014FEF8 0000000000000000
0000000014FF00 0000000000000000
0000000014FF08 0000000000000000
0000000014FF10 0000000000000000
0000000014FF18 0000000000000000
0000000014FF20 0000000000000000
0000000014FF28 0000000000000000
0000000014FF30 0000000000000000
0000000014FF38 0000000000000000
0000000014FF40 0000000000000000
0000000014FF48 0000000000000000
0000000014FF50 0000000000000000

"写K"=="ALLUSERSPROFILE=C:\\\\ProgramData"
返回到 hack.0000000140003629 自 hack.000000014004CD10
"simhei.ttf"
hack.0000000140003460
kernelbase.00007FFE6380000
hack.00000001400C0001

"ShooterGame "
"UnrealWindow"

"simhei.ttf"
"simhei.ttf"
"simhei.ttf"

"写K"=="ALLUSERSPROFILE=C:\\\\ProgramData"
返回到 hack.0000000140050065
"simhei.ttf"
"simhei.ttf"
"simhei.ttf"

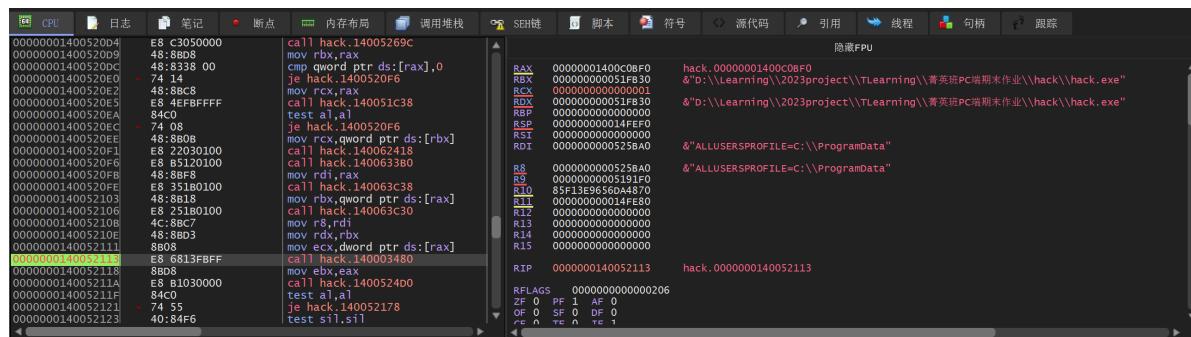
"写K"=="ALLUSERSPROFILE=C:\\\\ProgramData"
返回到 hack.0000000140052118 自 hack.0000000140003480
返回到 hack.0000000140052191 自 hack.00000001400525A4

返回到 kernel32.00007FFEE79F7344 自 ???

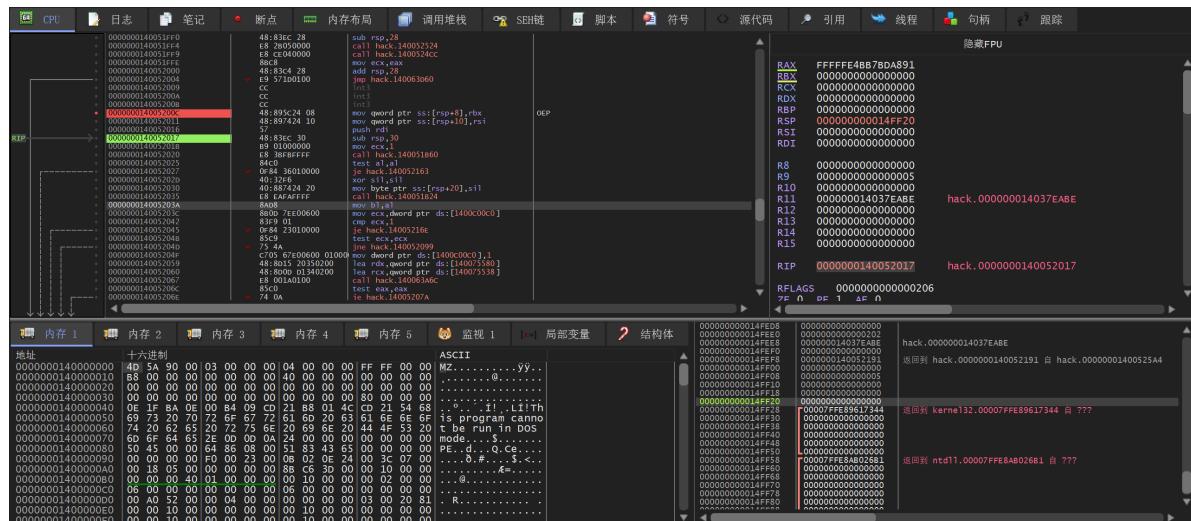
```

逐个查看调用链，可以发现 call hack.140003480 前有3个参数，分别是ecx = 1、rdx = "...\\hack.exe"，r8 = "...\\ProgramData"，调用ret后也有返回值eax。

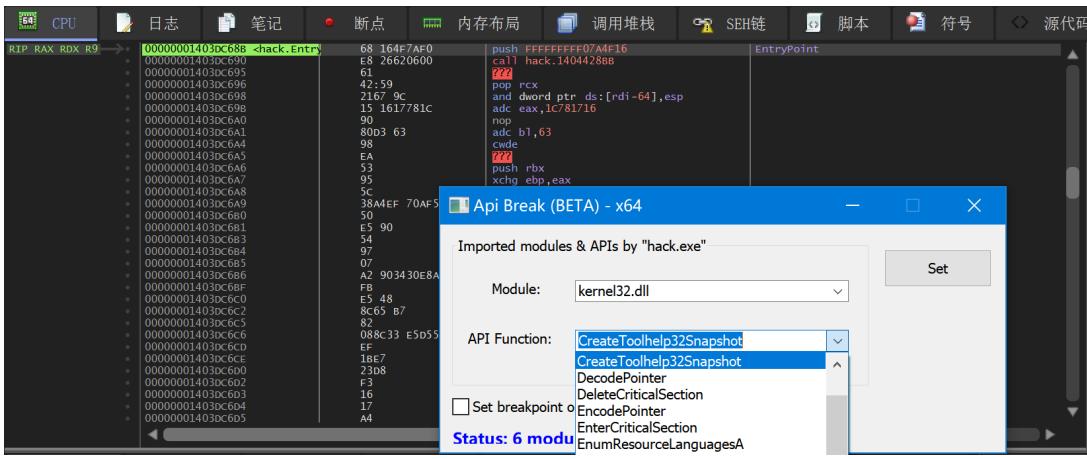
这是很明显的 int main(int argc, const char* argv[], const char* envp[]) 型调用，其中ecx表示命令行中只有一个字符串，rdx表示那一个字符串是hack.exe的路径，r8表示环境信息。



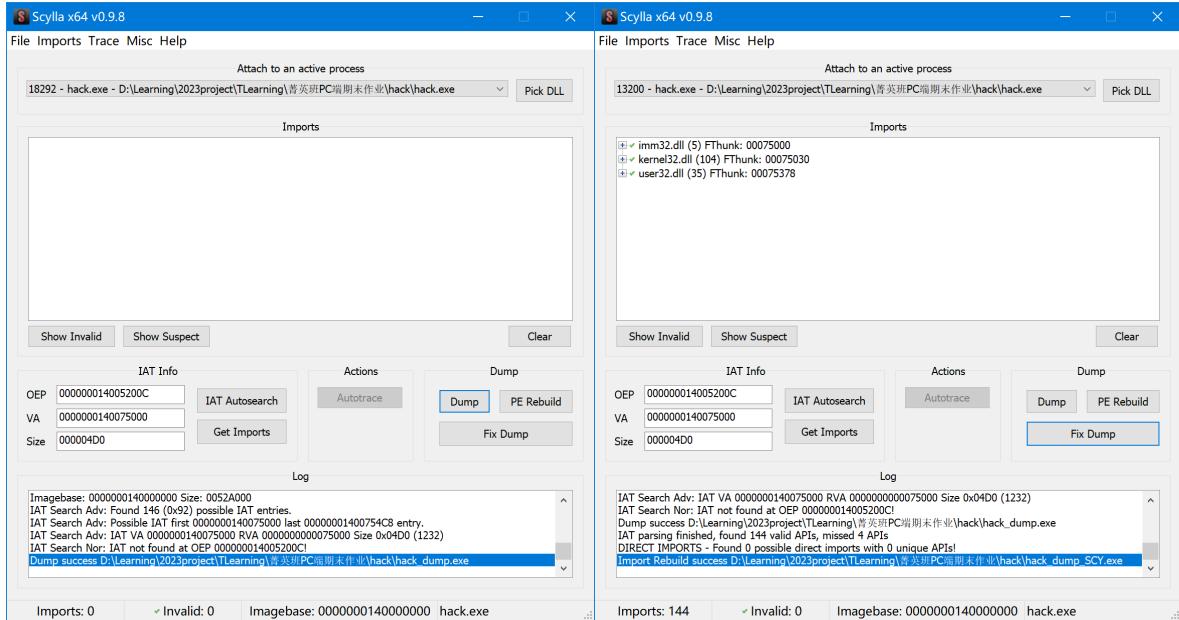
上述特征在其它call调用前是没有的，所以猜测hack.140003480在start层，那么向前回溯可以找到 public start OEP: 0x14005200C



另一种寻找OEP的方法是通过插件 Api Break 对指定内核系统调用下断点，这样就能进入程序分析调用栈，回溯拿到OEP，如 createToolhelp32Snapshot、GetModuleHandleA、OpenProcess 等注入程序常用的内核调用API，具体流程这里不赘述。



拿到OEP后使用插件 scylla x64 进行内存dump，再用Fix Dump将引用表导入修复文件，得到脱壳版 hack.exe即hack_dump_FIX.exe（插件默认起名为SCY，我改为FIX）。



2.2 hack_dump_FIX.exe逆向分析

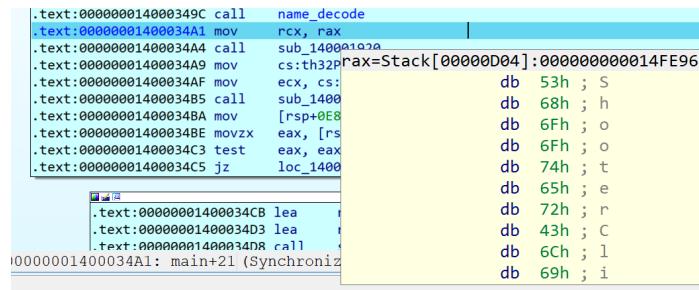
用IDA Pro直接打开hack_dump_FIX.exe，直接分析 F5 反汇编后的main函数。（下图是经过我逆向分析后的效果）

```

22 int64 v23; // [rsp+50h] [rbp-98h]
23 char *v24; // [rsp+58h] [rbp-90h]
24 _int64 v25; // [rsp+60h] [rbp-88h]
25 _int64 v26; // [rsp+68h] [rbp-80h]
26 char v27[11]; // [rsp+70h] [rbp-78h] BYREF
27 char v28[13]; // [rsp+7Bh] [rbp-6Dh] BYREF
28 char v29[14]; // [rsp+88h] [rbp-60h] BYREF
29 char v30[18]; // [rsp+96h] [rbp-52h] BYREF
30 char v31[24]; // [rsp+A8h] [rbp-40h] BYREF
31 char v32; // [rsp+C0h] [rbp-28h] BYREF
32
33 v3 = load1((__int64)&v13, v30); // ShooterClient.exe
34 v4 = decode1((__int64)v3); // ShooterClient.exe
35 th32ProcessID = GetPIDByName(v4); // 得到进程ID
36 if ( Process_is_existing(th32ProcessID) ) // 查看进程是否存在
37 {
38     v6 = load2(&v14, v29); // ShooterGame
39     lpWindowName = (LPCSTR)decode2(v6);
40     v7 = load3(&v15, v28); // UnrealWindow
41     lpClassName = (LPCSTR)decode3(v7);
42     hWnd = FindWindowA(lpClassName, lpWindowName);
43     GetClientRect(hWnd, &Rect); // 获取Client的Rect形状坐标
44     v8 = load1((__int64)&v16, v31); // ShooterClient.exe
45     v9 = decode1((__int64)v8); // ShooterClient.exe
46     ShooterClient_Base = (__int64)GetBaseByName(v9); // 查找模块基址
47     v18 = X_malloc(0x0000000000000000); // 分配一块内存, 后面应该是用于窗口
48     if ( v18 )
49         v19 = mem_init((__int64)v18); // 初始化内存
50     else
51         v19 = 0164;
52     v23 = v19;
53     HackWindowPtr = v19;
54     *(__DWORD *)(<v19 + 16) = Rect.right - Rect.left; // 窗口的长
55     *(__DWORD *)(<HackWindowPtr + 20) = Rect.bottom - Rect.top; // 窗口的宽
56     v24 = &v32;
57     v10 = load4(&v17, v27); // simhei.ttf, 网上查了是一种字体
58     v11 = decode4(v10); // simhei.ttf
59     v25 = set_font((__int64)v24, v11); // simhei.ttf
60     v26 = v25;
61     CreateHackWindow(HackWindowPtr, v25, ( __int64)Hacking); // 这里的sub_140003460是一个函数, 猜测和hack有关
62 }
63 sub_1400581EC(v5);
64 return 1;
65 }
```

可以看到外挂读写游戏内存主要通过 `GetPIDByName`、`FindWindowA`、`GetClientRect`、`GetBaseByName` 等函数得到游戏进程ID、窗口句柄和进程基址等。

如图，程序中的所有字符串都被加密，所以必须通过 `loadx()`; `decodeX()`; 两个函数解码才能得到字符串信息，但是我们可以直接通过动态调试得到明文（如下图所示）



下图是解密逻辑，其实挺简单的，如果没有动态调试也能直接破解：

```

1 int64 __fastcall sub_140003640( __int64 a1, __int64 a2)
2{
3     __int64 result; // rax
4     int i; // [rsp+0h] [rbp-18h]
5
6     for ( i = 0; i < 18; ++i )
7     {
8         *(_BYTE *)(a1 + i) = (i % 58 + 56) ^ *(_BYTE *)(a2 + i);
9         result = (unsigned int)(i + 1);
10    }
11    return result;
12}
```

知道字符串变量的明文后，看这些伪代码就和看源码差不多了。

首先是 `th32ProcessID = GetPIDByName(v4)`，进入函数 `GetPIDByName(v4)` 可以发现两个熟悉的内核调用，他们表示在进程快照中迭代遍历模块表，从而找到指定的模块基址。

```

1BYTE * __fastcall sub_140001AA0(__int64 a1)
2{
3    char *v1; // rax
4    __int64 v2; // rcx
5    unsigned __int8 v3; // dl
6    int v4; // eax
7    HANDLE hSnapshot; // [rsp+20h] [rbp-258h]
8    MODULEENTRY32 me; // [rsp+30h] [rbp-248h] BYREF
9
10   hSnapshot = CreateToolhelp32Snapshot(0x18u, th32ProcessID); // 创建进程快照
11   if ( hSnapshot == (HANDLE)-1i64 )
12       return 0i64;
13   me.dwSize = 568;
14   if ( !Module32First(hSnapshot, &me) )
15       return 0i64;
16   while ( 1 ) // 遍历模块列表, 寻找指定模块基址
17   {
18       v1 = me.szModule;
19       v2 = a1 - (_QWORD)me.szModule;
20       while ( 1 )
21       {
22           v3 = *v1;

```

其次是 ShooterClient_Base = (__int64)GetBaseByName(v9) 找到了 ShooterClient.exe 模块的基址，这在后续访问UWorld下的子类起重要作用。（内部函数的逻辑和上图完全一致，这里就不列出图片了。）

最后是一个类似malloc的函数分配了一块内存，同时设定了窗口的长、宽和字体类型simhei.ttf，可以推断这段代码最后的函数CreateHackWindow()维护了一个外挂的菜单窗口HackWindow。传入的3个参数分别是内存指针HackWindowPtr，格式结构体v25和一个函数指针Hacking。

```

v18 = X_malloc(0xC0ui64); // 分配一块内存, 后面应该是用于窗口
if ( v18 )
    v19 = mem_init((__int64)v18); // 初始化内存
else
    v19 = 0i64;
v23 = v19;
HackWindowPtr = v19;
*(__DWORD *)v19 + 16 = Rect.right - Rect.left; // 窗口的长
*(__DWORD *)HackWindowPtr + 20 = Rect.bottom - Rect.top; // 窗口的宽
v24 = &v32;
v10 = load4(v17, v27);
v11 = decode4(v10); // simhei.ttf, 网上查了是一种字体
v25 = set_font((__int64)v24, v11);
v26 = v25;
CreateHackWindow(HackWindowPtr, v25, (__int64)Hacking); // 这里的sub_140003460是一个函数, 猜测和hack有关

```

进入CreateHackWindow函数，发现只是普通的窗口更新函数，并没有明显的与作弊相关的代码，但其中有一处使用了传入的Hacking指针。它影响了每次窗口的更新，故CreateHackWindow中传入的函数指针Hacking应该会是外挂实现的主要逻辑。

```

do
{
    while ( PeekMessageA(&Msg, 0i64, 0, 0, 1u) )
    {
        TranslateMessage(&Msg);
        DispatchMessageA(&Msg);
        if ( Msg.message == 18 )
            v12 = 1;
    }
    if ( v12 )
        break;
    qmemcpy(v14, (const void *)sub_140004D60((unsigned int)&v40, v8, v9, v10, 0), sizeof(v14));
    sub_140045260();
    sub_140045900();
    sub_14000FB10();
    v13 = 1;
    if ( *(_QWORD *)(a1 + 24) ) // 这里使用了传入的函数指针
    {
        v29 = *(__int64 (__fastcall **)(__QWORD, __QWORD))(a1 + 24);
        v13 = v29(*(unsigned int *)(a1 + 16), *(unsigned int *)(a1 + 20));
    }
    if ( *(_BYTE *)a1 )
    {
        if ( !byte_1400C1388 )
        {
            sub_14004D410(a1, 0i64);
            byte_1400C1388 = 1;
        }
    }
}

```

进入Hacking函数，里面有两个函数，我在后期CE调试ShooterGame后推断出了它们的具体作用，但当前只能初步分析。

IDA View-A Pseudocode-B Pseudocode-A Hex View-1 Structures Enums

```

1 char Hacking()
2 {
3     ShowMenu(); // sub_140001B50, 可以推断是一个显示ui的模块函数
4     Cheating(); // sub_140003430
5     return 1;
6 }

```

进入第一个函数，`GetAsyncKeyState`是一个热键检查调用，网上查了数值 36 对应了键盘上的 Home 键，整个外挂程序只有一个地方使用的 Home 键，就是进入外挂后的那个窗口菜单，故显然这里是一个初始化 `HackMenu` 外挂菜单的逻辑。

```

20
21 if ( (GetAsyncKeyState(36) & 1) != 0 ) // 热键检查36, 是Home键, 控制gui显示与否
22     *(_BYTE *)HackWindowPtr = *(_BYTE *)HackWindowPtr == 0;
23 result = *(unsigned __int8 *)HackWindowPtr;
24 if ( *(_BYTE *)HackWindowPtr )
25 {
26     v10 = 298;
27     v1 = sub_140001680((float *)v14, *(float *)&dword_1400B9308, *(float *)&dword_1400B92FC);
28     sub_14001A820(v1, 4i64);
29     v11 = HackWindowPtr;
30     v2 = load5((__int64)&v7, v15);
31     v12 = decode5((__int64)v2); // "HOME 开关菜单",0
32     sub_140016370(v12, v11, v10);
33     v3 = load6(&v8, v13);
34     v4 = decode6(v3); // "透视",0
35     sub_14004AF20(v4, &byte_1400BFA80);
36     v5 = load7((__int64)&v9, v16);
37     v6 = decode7(v5); // "自瞄 (鼠标右键)",0
38     sub_14004AF20(v6, &byte_1400BFA81);
39     result = sub_1400196F0();
40 }
41 return result;
42 }

```

进入第二个函数的其中一个子函数，可以看到从 `ShooterClient_Base` 开始往下都是对某一个地址+偏移再传入一个函数，而函数中都有内存读取和内存复制的函数调用，故可知这些都是对游戏引擎中一些重要类对象的读取。

IDA View-A Pseudocode-A Hex View-1 Structures Enums Imports Exports

```

4 __int16 v2; // cx
5 __int16 _CX; // cx
6 const void *v4; // rax
7 __int16 v5; // cx
8 char v6; // r10
9 __int16 _CX; // cx
10 char v10[12]; // [rsp+24h] [rbp-34h] BYREF
11 char v11[40]; // [rsp+30h] [rbp-28h] BYREF
12
13 qword_1400C1280 = ((__int64 (__fastcall *)(__int64, _QWORD))sub_1400042F0)(ShooterClient_Base + 0x2F71060, 0i64);
14 qword_1400C12A0 = ((__int64 (__fastcall *)(__int64, _QWORD))sub_1400042F0)(ShooterClient_Base + 48685248, 0i64);
15 qword_1400C12C8 = ((__int64 (__fastcall *)(__int64, _QWORD))sub_1400042F0)(qword_1400C1280 + 48, 0i64);
16 qword_1400C12E8 = ((__int64 (__fastcall *)(__int64, _QWORD))sub_1400042F0)(qword_1400C12C8 + 152, 0i64);
17 dword_1400C129C = sub_140004320(qword_1400C12C8 + 160, 0i64);
18 qword_1400C1290 = ((__int64 (__fastcall *)(__int64, _QWORD))sub_1400042F0)(qword_1400C1280 + 352, 0i64);
19 __asm { rcl cx, 0EFh }
20 qword_1400C1288 = ((__int64 (__fastcall *)(__int64, _QWORD))sub_1400042F0)(qword_1400C1290 + 56, 0i64);
21 v1 = ((__int64 (__fastcall *)(__int64, _QWORD))sub_1400042F0)(qword_1400C1288, 0i64);
22 qword_1400C1278 = ((__int64 (__fastcall *)(__int64, _QWORD))sub_1400042F0)(v1 + 48, 0i64);
23 qword_1400C12B8 = ((__int64 (__fastcall *)(__int64, _QWORD))sub_1400042F0)(qword_1400C1278 + 944, 0i64);
24 _CX = v2 & ~1 << qword_1400C12B8;
25 __asm { rcl cl, 0A9h };
26 qword_1400C12B0 = ((__int64 (__fastcall *)(__int64, _QWORD))sub_1400042F0)(qword_1400C1278 + 968, 0i64);
27 qmemcpy(&unk_1400C1300, (const void *)sub_140004350(v10, qword_1400C12B0 + 6704, 0i64), 0xCui64);
28 v4 = (const void *)sub_140004350(v11, qword_1400C12B0 + 6716, 0i64);
29 qmemcpy(&unk_1400C1310, v4, 0xCui64);
30 LOBYTE(v5) = 0;
31 HIBYTE(v5) = BYTE1(v4);
32 _CX = (v5 + 1979) | (1 << v6);
33 __asm { rcl cl, 35h }
34 dword_1400C1298 = sub_1400043A0(qword_1400C12B0 + 6728, 0i64);
35 return qword_1400C12B8 && qword_1400C1278 && qword_1400C12E8 && dword_1400C129C;

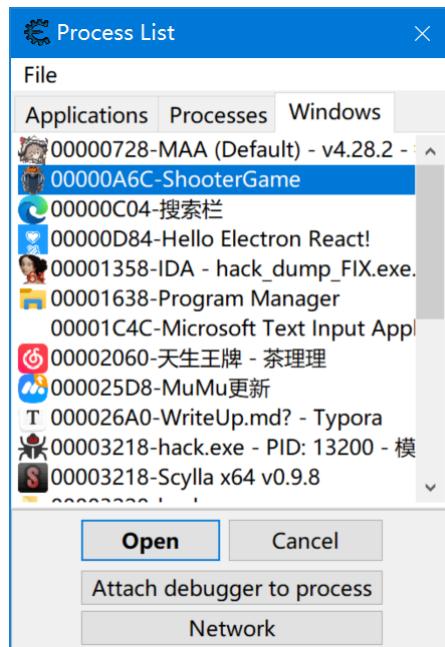
```

看到这里，就必须要知道这些偏移地址究竟代表着哪些类，故下面开始分析 `ShooterGame`。

3. 游戏内存初步分析

2.1 用CE进行游戏内容的简单分析

用CheatEngine挂载ShooterGame进程，尝试按照TLearning视频课中的方式，找到一些与游戏数据相关的内存。



进入游戏，可以看到手上的枪械有50发子弹，故先在CE中筛选值为 50 的内存。



Address	Value	Previous	First
dbghelp.dll+139600	50	50	50
E5BBC9C348	50	50	50
E5BBC9C5A8	50	50	50
E5BBC9C5D8	50	50	50
E5BBC9C5E8	50	50	50
E5BBC9C758	50	50	50
E5BBC9CCD8	50	50	50
E5BBC9D0A8	50	50	50
E5BBC9D0B8	50	50	50
E5BBC9D108	50	50	50
E5BBC9D3E8	50	50	50
E5BBC9D3F8	50	50	50
E5BBC9D448	50	50	50
E5BBC9DBC8	50	50	50
E5BBC9DBD8	50	50	50
E5BBC9DC28	50	50	50
E5BD71B88C	43	50	50
E5BDDBBDAE0	50	50	50

开枪，子弹还剩39发，使用 Next Scan 筛选出值为 39 的内存地址。



Found:1

Address	Value	Previous	First
2DC8723D284	39	39	50

New Scan Next Scan Undo Scan Settings

Value: Hex 39

Scan Type Exact Value Value Type 4 Bytes

Compare to first scan Lua formula
 Memory Scan Options Not
 All Unrandomizer
 Start 0000000000000000 Executable
 Stop 00007fffffff Enable Speedhack
 Writable CopyOnWrite
 Active memory only Alignment
 Fast Scan 4 Last Digits
 Pause the game while scanning

Memory View Add Address Manually

如上图，只剩一块内存，说明 [0xDC8723D284] 存放了子弹数量这个游戏关键信息。

Found:5

Address	Value	Previous	First
205276C6718	43FA01F9	42D807E6	42D807E8
205276C67C8	43FA01F9	42D807E6	42D807E8
205336C5A38	43FA01F9	42D807E6	42D807E8
205379F8A3C	4401C1D8	42FE0EC4	42FE0EC6
20539108518	43FA01F9	42D807E6	42D807E8

New Scan Next Scan Undo Scan Settings

Scan Type Changed value Value Type 4 Bytes

Compare to first scan Unrandomizer
 Memory Scan Options Enable Speedhack
 All
 Start 0000000000000000 Executable
 Stop 00007fffffff
 Writable CopyOnWrite
 Active memory only Alignment
 Fast Scan 4 Last Digits
 Pause the game while scanning

Memory View Add Address Manually

Active Description	Address	Type	Value
No description	1F799CD2664	4 Bytes	??
	1F79D5050684	8 Bytes	??
No description	1F79D25C5C4	8 Bytes	??
No description	1F79D86C5C4	8 Bytes	??
playerObject	1F7A63C0100	8 Bytes	??
bullet	1F7A63C0684	4 Bytes	??

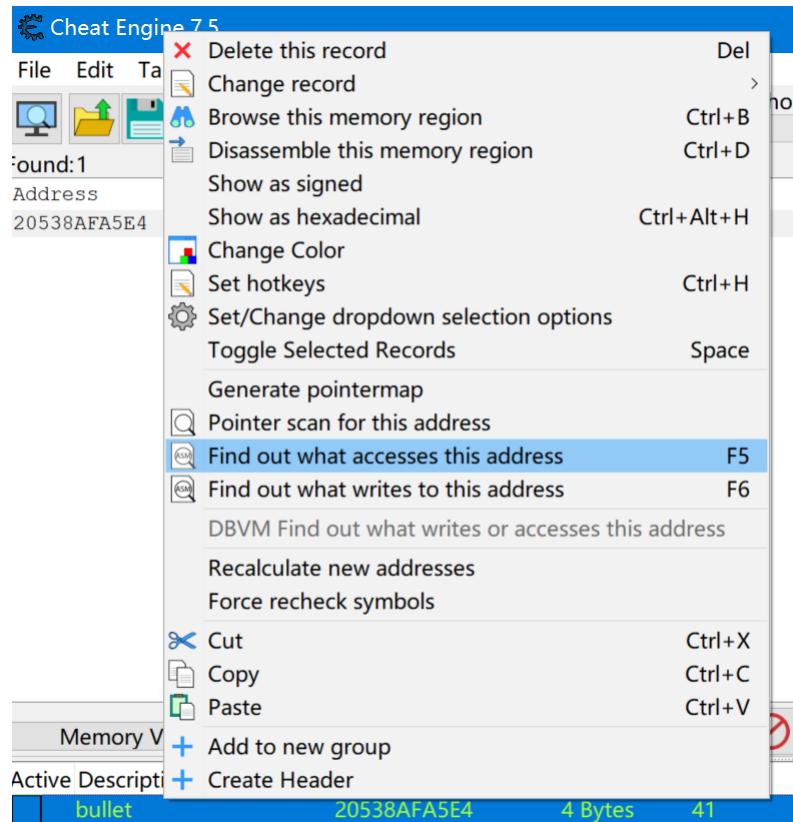
Advanced Options Table Extras

通过类似的方法（就是反复的Scan `changed value` 和 `unchanged value`）我们也能在上图中确定角色坐标的位置（这里的坐标是用浮点数存储的，所以不能直观看出结果）。

但是可以发现每次重开游戏或者角色重生后，这个地址就和子弹数量没有关系了，这说明了游戏中角色对象的内存分配不会是静态的，而是动态的，故如果想要永久地访问关键内存，必须要获取到调用这个信息的指针链，下面开始分析指针链。

2.2 UE4.22游戏指针链分析

首先可以通过调试器来找到子弹所处的类地址。



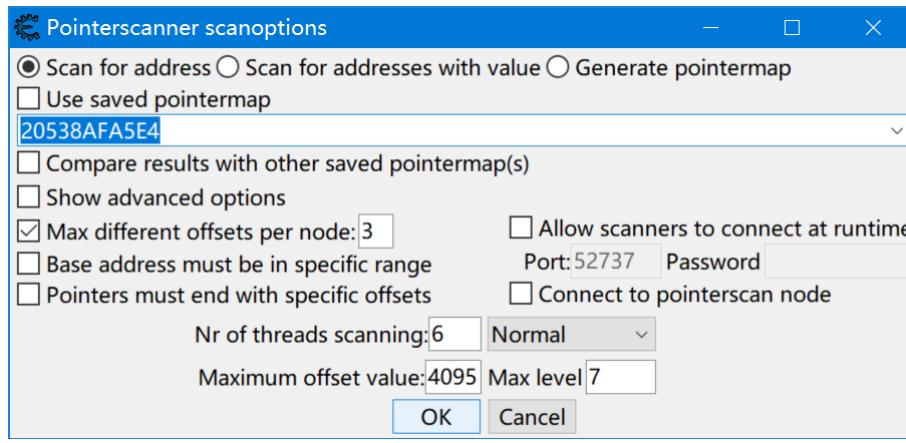
以我分析出来的其中一个地址 bullet = [0x20538AFA5E4] 为例，用 F5 调试器找到访问这个地址的汇编代码。

The screenshot shows the 'Accesses' window in Cheat Engine. It displays the assembly code for the address 20538AFA5E4. The window title is 'The following opcodes accessed 20538AFA5E4'. The assembly code listed is:

Count	Instruction
5174	7FF787753F63 - 8B 97 84050000 - mov edx,[rdi+00000584]
5174	7FF787754135 - 2B 97 84050000 - sub edx,[rdi+00000584]
51740	7FF787754330 - 66 0F6E 87 84050000 - movd xmm0,[rdi+00000584]

A context menu is open on the right side of the window, with options: Replace, Show disassembler, Add to the codelist, More information, and example.

通过 [rdi+0x584] 可以推断该对象的首地址为 $0x20538AFA5E4 - 0x584 = 0x20538AFA060$ ，这个地址就很可能是 AActor 角色对象的地址。但是像这样一层一层找是不现实的，故我们使用 Pointer Scan 功能进行指针链查找。



之前在IDA Pro中的逆向分析给了我参考，因为13和14行我已经确定了ShooterClient_Base基地址，而且也确定外挂所需要的两个偏移为0x2F71060和0x2E6E0C0故可以直接在bullet.PTR中进行筛选。

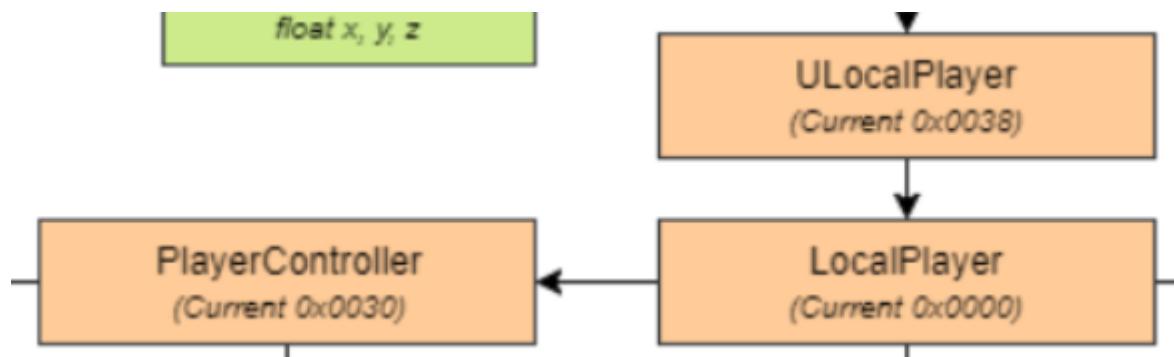
```

● 13 UWorld = ((__int64 (__fastcall *)(__int64, _QWORD))ReadQwordMem)(ShooterClient_Base + 0x2F71060, 0i64);
● 14 GName = ((__int64 (__fastcall *)(__int64, _QWORD))ReadQwordMem)(ShooterClient_Base + 0x2E6E0C0, 0i64);
● 15 ULevel = ((__int64 (__fastcall *)(__int64, _QWORD))ReadQwordMem)(UWorld + 48, 0i64);
● 16 ActorArray = ((__int64 (__fastcall *)(__int64, _QWORD))ReadQwordMem)(ULevel + 152, 0i64);
● 17 ActorCount = ((__int64 (__fastcall *)(__int64, _QWORD))ReadDwordMem)(ULevel + 160, 0i64);

```

Pointer paths:1053393							
Base Address	Offset 0	Offset 1	Offset 2	Offset 3	Offset 4	Offset 5	Offset 6
"ShooterClient.exe"+02F6E558	0	1C0	458	4F0	A68	130	584
"ShooterClient.exe"+02F6E558	0	1C8	468	4F0	A68	130	584
"ShooterClient.exe"+02F6E558	10	B8	30	70	8	130	55C
"ShooterClient.exe"+02F6E6E8	750	78	490	60	240	B8	584
"ShooterClient.exe"+02F6E6E8	DA8	498	78	60	240	B8	584
"ShooterClient.exe"+02F6E6E8	DB0	4A8	78	60	240	B8	584
"ShooterClient.exe"+02F6E6E8	D80	30	268	1F8	8	E8	584
"ShooterClient.exe"+02F6E6E8	BDO	0	268	1F8	8	E8	584
"ShooterClient.exe"+02F6E6E8	DA8	458	10	1F8	8	E8	584
"ShooterClient.exe"+02F6E6E8	DB0	468	10	1F8	8	E8	584
"ShooterClient.exe"+02F6E6E8	9C0	10	F0	1F8	8	E8	584
"ShooterClient.exe"+02F6E6E8	750	78	498	1F8	8	E8	584
"ShooterClient.exe"+02F6E6E8	DA8	4B0	A10	1F8	8	E8	584
"ShooterClient.exe"+02F6E6E8	DB0	4C0	A10	1F8	8	E8	584
"ShooterClient.exe"+02F6E6E8	CA0	750	78	1F8	8	E8	584
"ShooterClient.exe"+02F6E6E8	750	78	1F8	8	E8	584	20538AFA5E4 = 37
"ShooterClient.exe"+02F6E6E8	9A8	80	F0	1F8	8	E8	584
"ShooterClient.exe"+02F6E6E8	A30	10	98	60	388	E8	584
"ShooterClient.exe"+02F6E6E8	A30	0	30	60	388	E8	584
"ShooterClient.exe"+02F6E6E8	A30	10	110	120	388	E8	584
"ShooterClient.exe"+02F6E6E8	D80	30	268	1E8	F0	E8	584
"ShooterClient.exe"+02F6E6E8	BDO	0	268	1E8	F0	E8	584
"ShooterClient.exe"+02F6E6E8	DA8	458	10	1E8	F0	E8	584
"ShooterClient.exe"+02F6E6E8	DB0	468	10	1E8	F0	E8	584
"ShooterClient.exe"+02F6E6E8	9C0	10	F0	1E8	F0	E8	584
"ShooterClient.exe"+02F6E6E8	750	78	498	1E8	F0	E8	584
"ShooterClient.exe"+02F6E6E8	DA8	4B0	A10	1E8	F0	E8	584
"ShooterClient.exe"+02F6E6E8	DB0	4C0	A10	1E8	F0	E8	584
"ShooterClient.exe"+02F6E6E8	CA0	750	78	1E8	F0	E8	584

但可以看到过滤后的结果仍然非常多，所以需要得到一条确定的指针链。故通过网上学习相关UE4逆向笔记+IDA Pro对外挂汇编代码的分析，我们可以确定UWorld的子类中有一条ULocalPlayer -> LocalPlayer -> PlayerController的指针链，其中偏移为+0x38 -> +0x0 -> +0x30。（如下图所示）



通过这条指针链我们也能逆向到前面的变量，即通过 `ULocalPlayer = [GameInstance + 0x38]` 确认了GameInstance变量。

```

● 20 ULocalPlayer = ((__int64 (__fastcall *)(__int64, _QWORD))ReadQwordMem)(GameInstance + 0x38); // "GameInstance + 0x38"
● 21 LocalPlayer = ((__int64 (__fastcall *)(__int64, _QWORD))ReadQwordMem)(ULocalPlayer, 0i64); // "ULocalPlayer + 0x00"
● 22 PlayerController = ((__int64 (__fastcall *)(__int64, _QWORD))ReadQwordMem)(LocalPlayer + 0x30, 0i64); // "LocalPlayer + 0x30"

```

再根据IDA中的逆向信息筛选指针链：

```

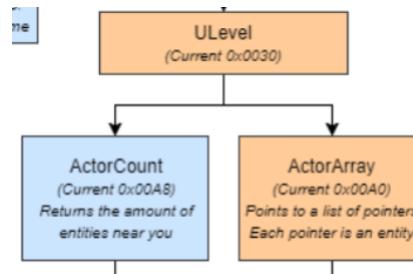
1 Object1 = [Base + 0x2F71060]
2 GameInstance = [Object1 + 0x160]
3 ULocalPlayer = [GameInstance + 0x38]
4 LocalPlayer = [ULocalPlayer + 0]
5 PlayerController = [LocalPlayer + 0x30]
6 Object2 = [PlayerController + 0x3B0]
7 Object3 = [PlayerController + 0x3CB]

```

Pointer paths:1053393							
Base Address	Offset 0	Offset 1	Offset 2	Offset 3	Offset 4	Offset 5	Offset 6
"ShooterClient.exe"+02F6EDB0	160	50	FF8	20	98	DF8	BE4
"ShooterClient.exe"+02F71060	160	38	0	30	3B0	778	584
"ShooterClient.exe"+02F71060	160	38	0	30	370	778	584
"ShooterClient.exe"+02F71060	160	38	0	30	360	778	584

上图是筛选出来的指针链，可以猜测指针链中的第一个对象其实是一个**UWorld**对象，而倒数两个未知的对象可能是**AActor**数组中的**角色基址 (ActorArray[0])**或者主玩家的角色地址。

下图显示IDA中剩下的两处未知指针链，由于已知 **Uworld** -> **ULevel** -> **ActorArray/ActorCount** 链，所以也能分析出未知对象名（图中的偏移与实际稍有不同，这可能是因为这张图是UE4.23版本，但影响不大）



```

● 13 UWorld = ((__int64 (__fastcall *)(__int64, _QWORD))ReadQwordMem)(ShooterClient_Base + 0x2F71060, 0i64);
● 14 GName = ((__int64 (__fastcall *)(__int64, _QWORD))ReadQwordMem)(ShooterClient_Base + 0x2E6E0C0, 0i64);
● 15 ULevel = ((__int64 (__fastcall *)(__int64, _QWORD))ReadQwordMem)(UWorld + 0x30, 0i64); // "World + 0x30"
● 16 ActorArray = ((__int64 (__fastcall *)(__int64, _QWORD))ReadQwordMem)(ULevel + 0x98, 0i64); // "Ulevel + 0xA0"
● 17 ActorCount = ((__int64 (__fastcall *)(__int64, _QWORD))ReadDwordMem)(ULevel + 0xA0, 0i64); // "Ulevel + 0xA8"

```

还有一个未知对象是 `[base + 0x2E6E0C0]`，其实这个对象在后文中也较少使用。不过在后面对遍历 **ActorArray** 时解码机器人名字的函数 `sub_1400020B0` 中找到了**GName** 的身影。调用链为 `sub_1400020B0 -> sub_140001E50 -> ReadQwordMem(8*(a2 / 0x4000) + GName)`。

```

BotActor = ReadQwordMem(ActorArray + 8 * i);
if ( !BotActor || BotActor == PlayerActor )// 确认是非玩家机器人
    goto LABEL_13;
v30 = sub_1400020B0((__int64)v45, BotActor); // 获取机器人name数据
v31 = v30;
v10 |= 1u;
v2 = load8(&v11, v42);
v28 = decode8(v2); // 解码机器人名字
v3 = (unsigned __int8 *)sub_140003A80(v31);

1 __int64 __fastcall sub_1400020B0(__int64 a1, __int64 a2)
2{
3     unsigned int v3; // [rsp+24h] [rbp-14h]
4
5     v3 = ReadDwordMem((const void *)(a2 + 24));
6     sub_140001E50(a1, v3);
7     return a1;
8}

```

因为游戏中的**字符串名称**是单独存放的，在其他对象中的玩家、角色对象都是用**ID号**指代，当我们需要获取角色的字符串名称时，就必须要访问一个全局类对象 `GName`，故这里的最后一个位置对象应该是 `GName`。

```

19     if ( GName )
20     {
21         v12 = ReadQwordMem(8 * (a2 / 0x4000) + GName);
22         if ( v12 )
23         {
24             v13 = ReadQwordMem(8 * (a2 % 0x4000) + v12);
25             if ( v13 )
26             {
27                 v17 = byte_14009C951;
28                 memset(v18, 0, sizeof(v18));
29                 sub_140001A10((const void *)(v13 + 12), &v17, 0x40ui64);
30                 set_font(a1, (_int64)&v17);
31             }
32         }
33     }

```

至此我们已经得到了分析外挂逻辑所需要的重要信息——存放游戏中重要对象内存地址的变量，最后对传入 CreateHackwindow() 函数的 Hacking() 函数进行总结性分析。

```

13 UWorld = ((__int64 (__fastcall *)__int64, _QWORD))ReadQwordMem)(ShooterClient_Base + 0x2F71060, 0i64);
14 GName = ((__int64 (__fastcall *)__int64, _QWORD))ReadQwordMem)(ShooterClient_Base + 0x2E6E0C0, 0i64);
15 ULevel = ((__int64 (__fastcall *)__int64, _QWORD))ReadQwordMem)(Uworld + 0x38, 0i64); // "World + 0x30"
16 ActorArray = ((__int64 (__fastcall *)__int64, _QWORD))ReadQwordMem)(ULevel + 0x98, 0i64); // "ULevel + 0xA0"
17 ActorCount = ((__int64 (__fastcall *)__int64, _QWORD))ReadDwordMem)(ULevel + 0xA0, 0i64); // "ULevel + 0xA8"
18 GameInstance = ((__int64 (__fastcall *)__int64, _QWORD))ReadQwordMem)(Uworld + 0x160, 0i64);
19 __asm { rcl cx, 0Fh }
20 ULocalPlayer = ((__int64 (__fastcall *)__int64, _QWORD))ReadQwordMem)(GameInstance + 0x38, 0i64); // "GameInstance + 0x38"
21 LocalPlayer = ((__int64 (__fastcall *)__int64, _QWORD))ReadQwordMem)(ULocalPlayer, 0i64); // "ULocalPlayer + 0x00"
22 PlayerController = ((__int64 (__fastcall *)__int64, _QWORD))ReadQwordMem)(LocalPlayer + 0x30, 0i64); // "LocalPlayer + 0x30"
23 PlayerActor = ((__int64 (__fastcall *)__int64, _QWORD))ReadQwordMem)(PlayerController + 0x3B0, 0i64);
24 _CX = v2 & ~1 << PlayerActor;
25 __asm { rcl cl, 09h }
26 PlayerBase = ((__int64 (__fastcall *)__int64, _QWORD))ReadQwordMem)(PlayerController + 0x3C8, 0i64);
27 qmemcpy(
28     &PlayerPos,
29     (const void *)(__int64 (__fastcall *)__char *, __int64, _QWORD))ReadPositionMem)(v10, PlayerBase + 0x1A30, 0i64),
30     0xCui64); // 这里一下读取3个值，显然是三维坐标
31 v4 = (const void *)(__int64 (__fastcall *)__char *, __int64, _QWORD))ReadPositionMem)(v11, PlayerBase + 0x1A3C, 0i64);
32 qmemcpy(&unk_1400C1310, v4, 0xCui64);
33 LOBYTE(v5) = 0;
34 HIBYTE(v5) = BYTE1(v4);
35 _CX = (v5 + 1979) | (1 << v6);
36 __asm { rcl cl, 35h }
37 PlayerPerspective = ReadFloatMem(PlayerBase + 6728, 0i64); // 浮点数，说明是玩家视角
38 return PlayerActor && PlayerController && ActorArray && ActorCount;
39
40 }

```

4. 外挂逻辑分析

显示外挂菜单gui的函数 ShowMenu() 我在前面已经分析完毕，因此本节主要分析作弊函数 Cheating()

```

1 char Hacking()
2 {
3     ShowMenu();
4     Cheating(); // sub_140001B50, 可以推断是一个显示ui的模块函数
5     return 1;
6 }

```

Cheating 中第一、三个函数中并没有对游戏内存对象的调用，又由于传入了窗口指针 HackWindowPtr，所以合理推断只是对窗口信息的更新而已。主要逻辑在第二个函数 CheatingMain() 中

```

1 unsigned __int64 Cheating()
2 {
3     sub_14004D4B0(HackWindowPtr); // 更新窗口信息
4     CheatingMain(); // 外挂作弊主要逻辑
5     return sub_14004D530(HackWindowPtr); // 更新窗口信息
6 }

```

CheatingMain() 函数的第一部分主要内容就是遍历ActorArray()数组，从中筛选出所有Ai自动控制的Bot，同时保留他们的名称字符串。

```

● 49 v9 = 0;
● 50 result = (unsigned __int8)ReadingMem();           // 读取内存中的对象地址
● 51 if ( (_BYTE)result )
● 52 {
● 53     v14 = *(float *)&const_float_400_0;           // 浮点常数400.0
● 54     v25 = 0i64;
● 55     for ( i = 0i64; i < (unsigned int)ActorCount; ++i )// 遍历ActorArray
● 56     {
● 57         BotActor = ReadQwordMem(ActorArray + 8 * i, 0i64);
● 58         if ( !BotActor || BotActor == PlayerActor )
● 59             goto is_not_ai;                           // 不是ai bot
● 60         v29 = GetName(v44, BotActor);               // 从GName中获取name
● 61         v30 = v29;
● 62         v9 |= 1u;
● 63         v1 = load8(&v10, v41);
● 64         v27 = decode8(v1);                         // 解码字符串
● 65         v2 = (unsigned __int8 *)sub_140003A80(v30);
● 66         v3 = v27 - (_QWORD)v2;
● 67         while ( 1 )                                // 这里v2明显是字符串首地址, v3是字符串长度
● 68         {
● 69             v4 = *v2;
● 70             if ( *v2 != v2[v3] )                     // 其其实现的就是strcmp, 检查名称是否匹配
● 71                 break;
● 72             ++v2;
● 73             if ( !v4 )
● 74             {
● 75                 v5 = 0;
● 76                 goto name_error;
● 77             }
● 78         }
● 79         v5 = v4 < v2[v3] ? -1 : 1;
● 80 name_error:
● 81     if ( v5 )
● 82 is_not_ai:
● 83     true_flag = 0;
● 84     else
● 85     true_flag = 1;
● 86     v8 = true_flag;

```

`CheatMain()` 的第二部分主要处理aiobot位置坐标和玩家视角坐标。首先读取player坐标和bot坐标，计算距离并单位换算，然后在使用 `HackWindowPtr` 显示框框之前，先做了一次 `CheckVision()` 的检查，确定可以在可视范围内显示后才进行显示。最后也计算了屏幕中心到bot的距离，这是为后面自瞄程序做准备的。

```

v25 = ReadQwordMem(BotActor + 344);
if ( v25 )
{
    ReadPositionMem(botPos_raw, (const void *)(v25 + 356));// 读取bot坐标
    qmemcpy(botPos, botPos_raw, 0xCui64);
    distance_between_player_and_bot = GetDistance(&playerPos, botPos); // 计算两点间距离
    distance_show = (int)(float)(*(float *)&distance_between_player_and_bot / *(float *)&const_float); // // 除以一个常数，可能是为了单位换算
    Zero_Perspective(&playerPerspective, 0.0, 0.0, 0.0); // 初始化了视角变量
    qmemcpy(botPos_tmp, botPos_raw, 0xCui64);
    if ( CheckVision((__int64)botPos_tmp, &playerPerspective) )// 检查目标在屏幕可视范围内
    {
        if ( cheat_vision_open )                  // 检查是否打开了透视挂
        {
            v7 = load9((__int64)v12, v43);      // 读取透视时显示的字符串信息“机器人%dm”
            v29 = decode9((__int64)v7);          // 解码透视时显示的字符串信息“机器人%dm”
            qmemcpy(v39, sub_1400018D0(v44, 255, 255, 255, 255), sizeof(v39));
            qmemcpy(v40, v39, sizeof(v40));       // 这里连续复制了3遍，黑人问号？？
            qmemcpy(v41, v40, sizeof(v41));
            LODWORD(distance_dword) = distance_show;
            sub_14004EAD0(HackWindowPtr, (int)playerPerspective, (int)v24, v41, 20, v29, distance_dword); // // 传入了HackWindowPtr和distance_show, 是为了显示透视距离
        }
        Zero_Perspective_1(v18);
        *(float *)v18 = playerPerspective - (float)((float)*(int *)(HackWindowPtr + 16) / *(float *)&const_float_1);
        *(float *)&v18[1] = v24 - (float)((float)*(int *)(HackWindowPtr + 20) / *(float *)&const_float_1); // // 计算量屏幕中心到目标的距离
        // 屏幕中心也是准星，应该是为自瞄准备数据
    }
}

```

下图是 `CheckVision()` 函数的返回值，有对 `HackWindowPtr+0x10` 和 `HackWindowPtr+0x14` 的调用。这明显是比较目标坐标是否超出了窗口的范围，如果超出范围透视的框框自然也无法在游戏中显示。

```

● 53 return *a2 > *(float *)&const_float_2
● 54     && (float)*(int *)(HackWindowPtr + 16) > *a2 // 比较窗口的长
● 55     && a2[1] > *(float *)&const_float_2
● 56     && (float)*(int *)(HackWindowPtr + 20) > a2[1]; // 比较窗口的宽
● 57 }

```

`CheatMain()` 最后一个部分是有关自瞄的外挂逻辑，在计算了视角中心到目标bot的距离后，每次都会进行一次比较，维护 `min_target` 和 `min_bot` 变量，在外挂逻辑中，`min_bot` 被作为 `targetBot` 与屏幕中心计算偏移距离，调用的是函数 `calcu_centroid_offset`，最后向 `PlayerController+920` 内存写入准心偏移，完成自瞄修改。

```

● 118     Zero_Perspective_1(v18);
● 119     *(float *)v18 = playerPerspective - (float)((float)*(int *)(&HackWindowPtr + 16) / *(float *)&const_float_1); // 长
● 120     *(float *)v18[1] = v24 - (float)((float)*(int *)(&HackWindowPtr + 20) / *(float *)&const_float_1); // 宽
● 121         // 计算量屏幕中心到目标的距离
● 122         // 屏幕中心也是准星，应该是为自瞄准备数据
● 123     v20 = sub_1400015B0();
● 124     v22 = v20 * sub_1400015B0();
● 125     v21 = sub_1400015B0();
● 126     sub_1400015B0();
● 127     distance_target = sqrt_a1(); // 最后调用sqrt，计算开根号后的直线距离
● 128     if ( min_target >= distance_target ) // 记忆化存储，保存了离准心最近的一名敌人
● 129     {
● 130         min_target = distance_target;
● 131         min_bot = botActor;
● 132     }
● 133 }
● 134 }
● 135 }
● 136 }
● 137 result = (unsigned __int8)cheat_scope_open;
● 138 if ( cheat_scope_open ) // 检查自瞄挂开了没
● 139 {
● 140     if ( !targetBot )
● 141         targetBot = min_bot; // 将距离准星最近的敌人作为自瞄目标
● 142     result = (unsigned __int16)GetAsyncKeyState(2) & 0x8000; // 这里是判断右键
● 143     if ( (_DWORD)result )
● 144     {
● 145         if ( targetBot )
● 146         {
● 147             result = ReadQwordMem(targetBot + 344);
● 148             v27 = result;
● 149             if ( result )
● 150             {
● 151                 ReadPositionMem(botPos_taget, (const void *)(v27 + 356)); // 读取坐标
● 152                 qmemcpy(v36, botPos_taget, 0xCui64);
● 153                 qmemcpy(v37, &playerPos, 0xCui64);
● 154                 Calcu_centroid_offset(distance_between_aim_and_target, v37, v36); // 计算准心偏移
● 155                 result = Write_Aim(PlayerController + 920, distance_between_aim_and_target); // 写入准心偏移

```

下面是写入准心偏移的主要依赖调用，其实就是利用 `WriteProcessMemory` 向进程句柄直接写入内存

```

IDA View-A Pseudocode-A Structures Enum
1 BOOL __fastcall sub_140001A60(void *a1, const void *a2, SIZE_T a3)
2 {
3     SIZE_T NumberOfBytesWritten[3]; // [rsp+30h] [rbp-18h] BYREF
4
5     return WriteProcessMemory(hProcess, a1, a2, a3, NumberOfBytesWritten);
6 }

```

5. 总结

综上，我先使用x64dbg和IDA Pro，通过反调试、解混淆、Dump内存找到外挂程序的关键代码，通过初步分析，我了解了外挂读写进程内存的方式和**修改进程内存**、**附加窗口信息**的方式。我确定了需要明确的游戏**关键逻辑**和**指针链**。

进而我用Cheat Engine分析游戏内存，通过附加调试器、查找指针链确定了游戏关键逻辑类对象的相对关系。

最后我回到IDA中逆向外挂程序，通过分析作弊主要函数逆向出了透视、自瞄外挂的主要实现原理，该程序读取非玩家机器人信息，并计算其与玩家相对位置，再分别将透视字符串通过进程窗口句柄写入窗口、通过 `GameInstance` 对象将自瞄准心偏移写入游戏内存。